Reference Counting

Kevin Köster

Uni Hamburg

31. März 2013

Frage

Wann wissen wir, ob ein Objekt freigegeben werden kann?

```
1 void foo() {
    char* data = malloc(1024);
3
    calc_unblocked(data);
5
    calc_blocked(data);
7
    //free?!?
9 }
```

Möglichkeiten

- Garbage Collection
- Reference Counting

Garbage Collection

```
1 public static void foo(){

3   ArrayList<String> liste = new ArrayList<String>(); //
        heap

5   bar(liste);
}
```

Listing 1: GC

Garbage Collection

Eigenschaften

Überwacht alle Objekte und löscht diese, wenn nötig.

Garbage Collection

Vorteile

- Einfach (Benutzer)
- Sicher

Nachteile

- Aufwändige Implementierung
- Kostet Rechenleistung
- Zufällig

Reference Counting

Eigenschaften

Zählung der Referenzen auf ein bestimmtes Objekt.

Reference Counting

Vorteile

- Leichte Implementierung
- Schnell

Nachteile

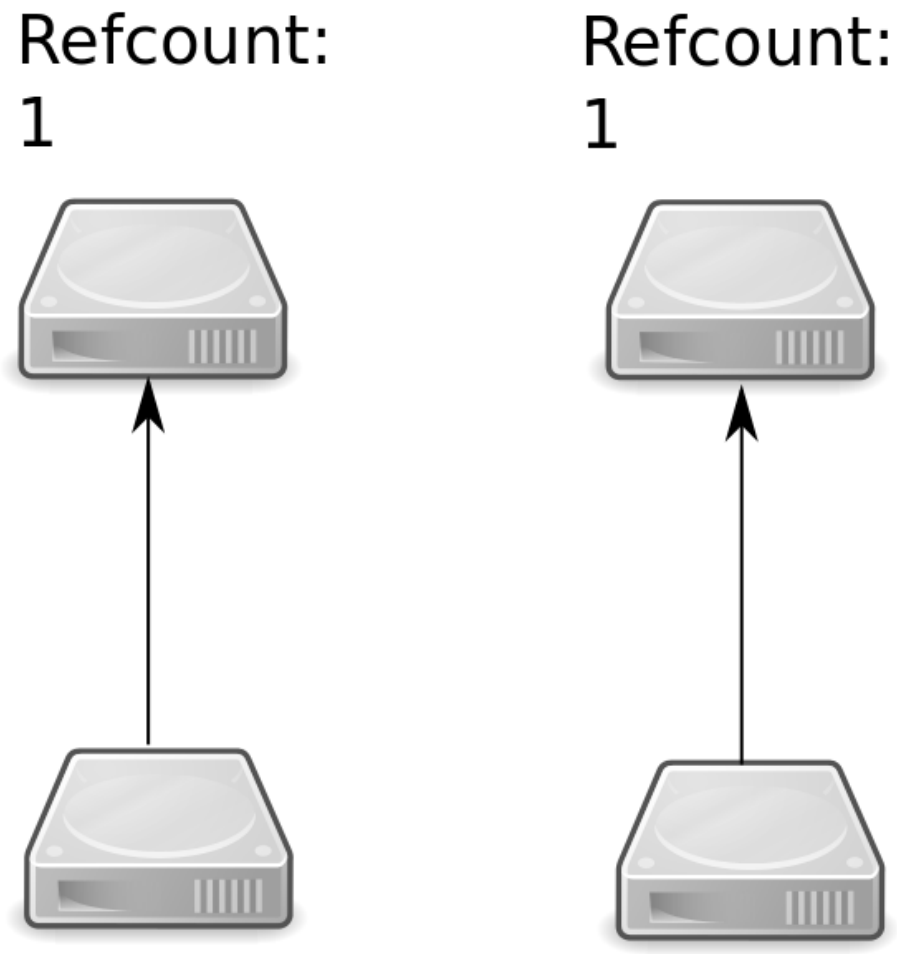
- Aufwändig für Benutzer
- Fehleranfällig
- Zyklen

Zyklen

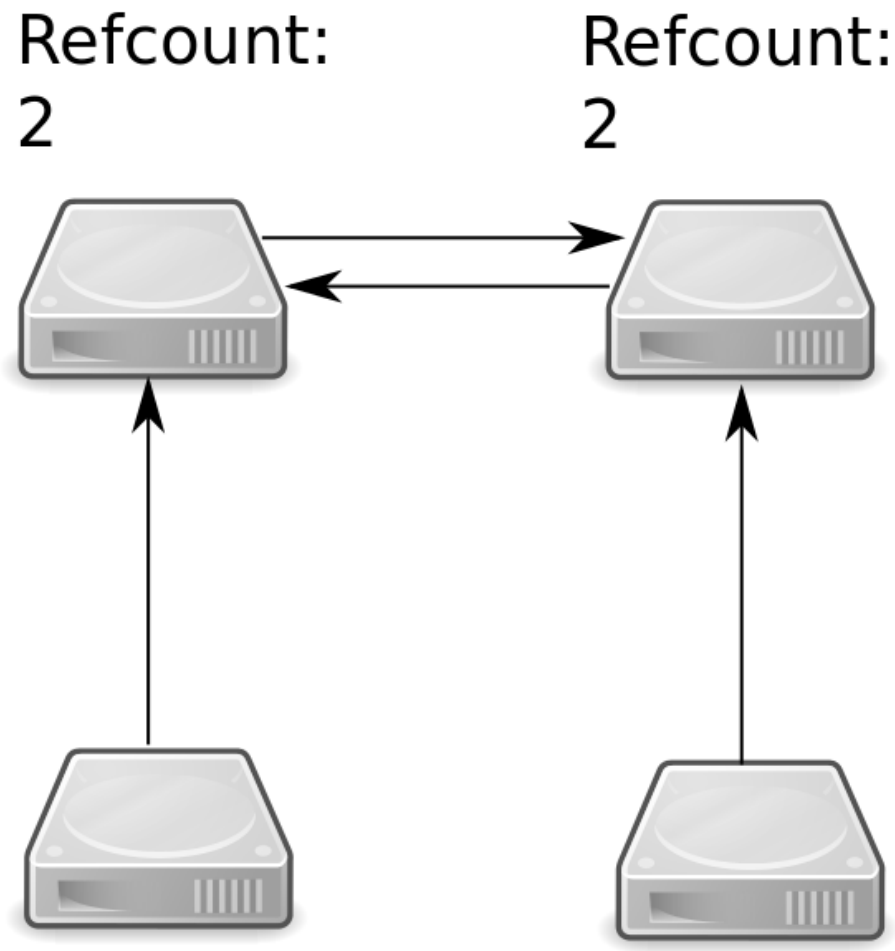
Problem

Zwei Objekte referenzieren sich gegenseitig

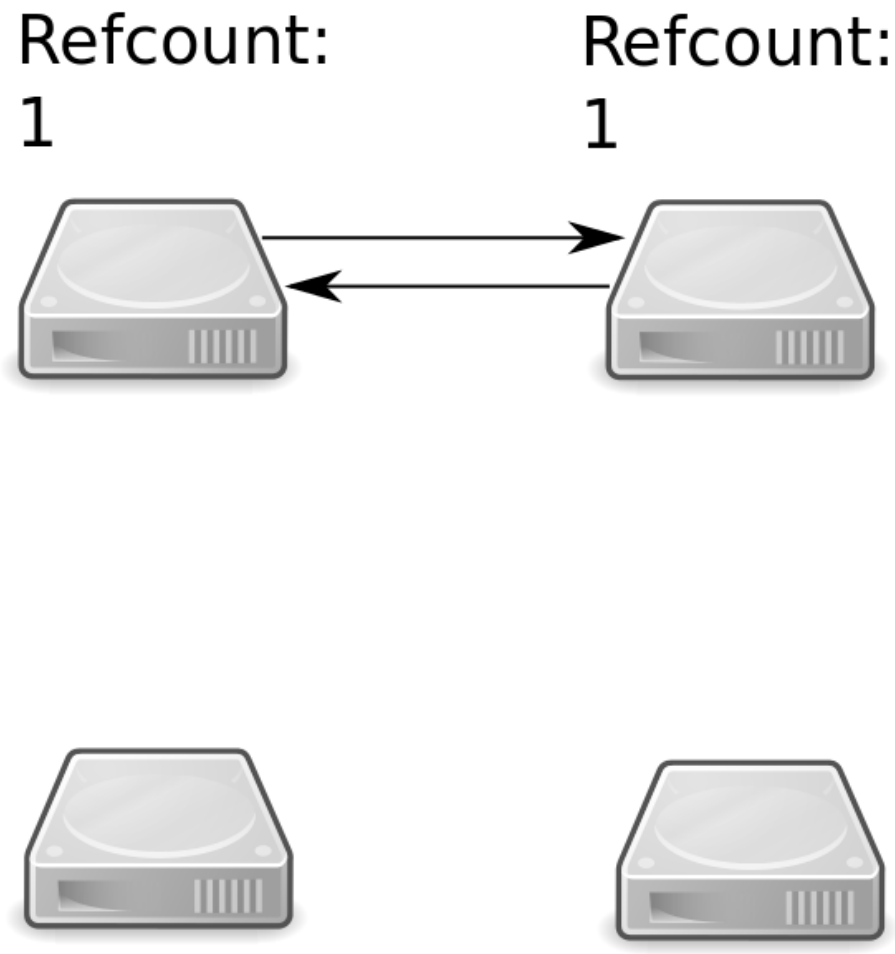
Zyklen



Zyklen



Zyklen

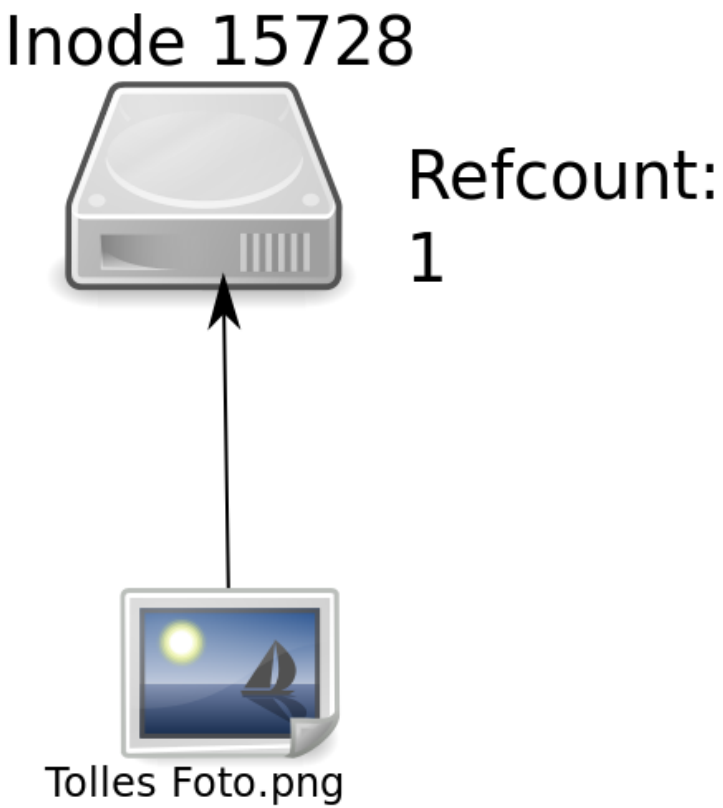


Inode

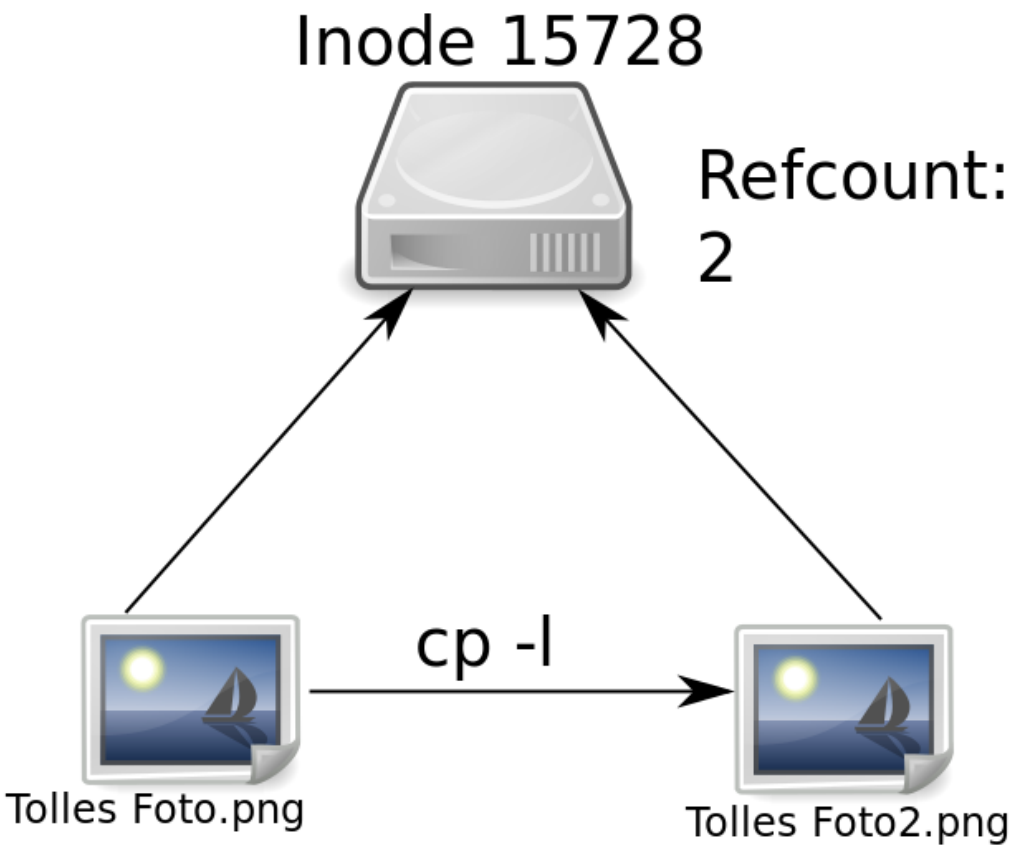
Inhalt eines Inode

- Zugriffsrechte
- Eigentümer und Gruppe (UID und GID)
- Dateityp
- Größe
- Änderungsdaten
- Referenzzähler

Inode



Inode

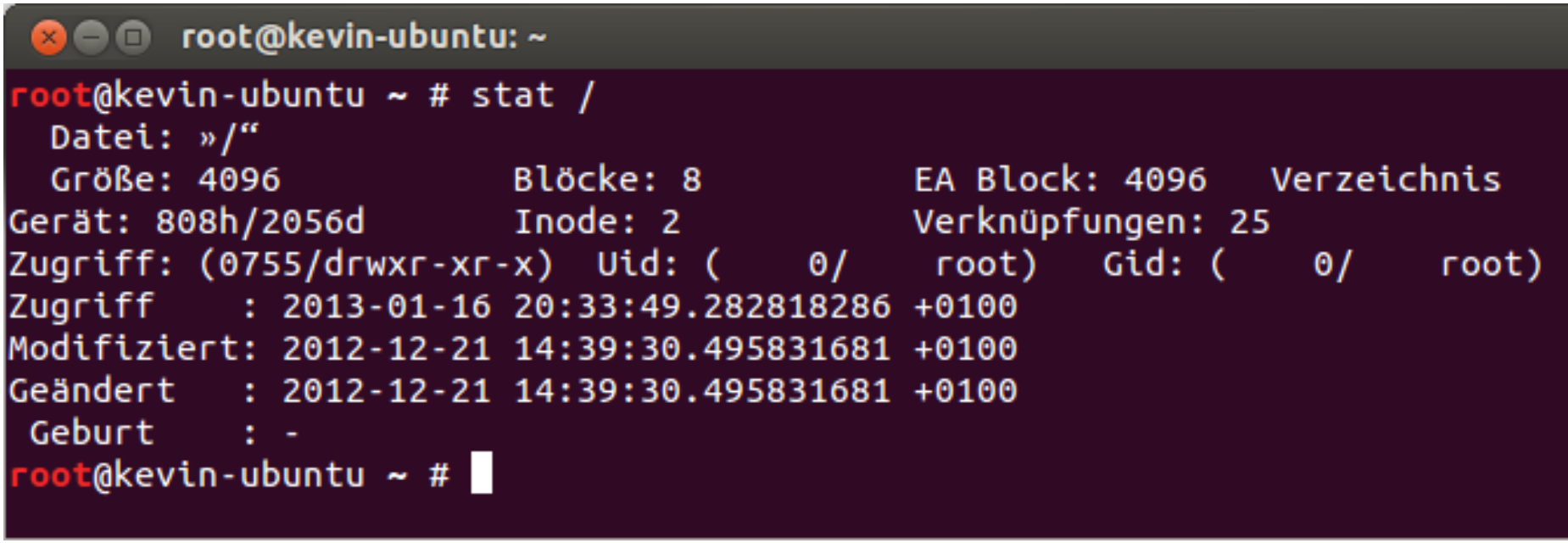


Inode

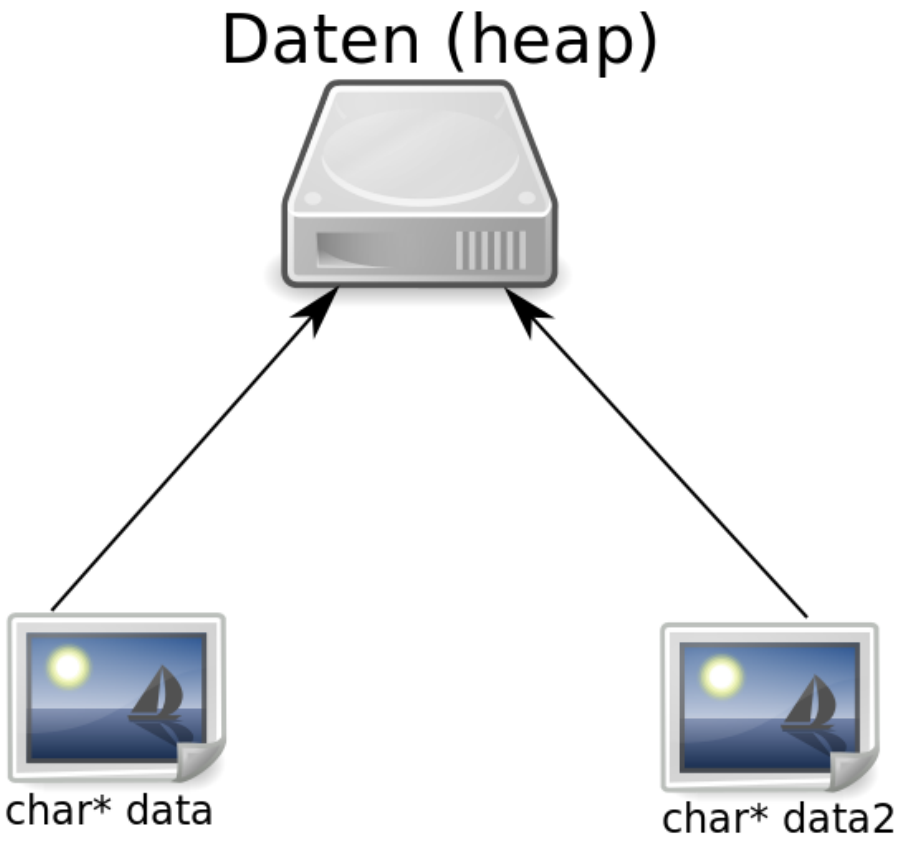
Referenzzähler

- Erhöht sich für jede Referenz um 1
- Jeder Hardlink eine Referenz (Nicht unterscheidbar)
- Kein direktes Löschen nur unlink

stat


root@kevin-ubuntu: ~
root@kevin-ubuntu ~ # stat /
  Datei: »/“
  Größe: 4096   Blöcke: 8   EA Block: 4096   Verzeichnis
Gerät: 808h/2056d   Inode: 2   Verknüpfungen: 25
Zugriff: (0755/drwxr-xr-x) Uid: (0/   root)   Gid: (0/   root)
Zugriff   : 2013-01-16 20:33:49.282818286 +0100
Modifiziert: 2012-12-21 14:39:30.495831681 +0100
Geändert   : 2012-12-21 14:39:30.495831681 +0100
 Geburt   : -
root@kevin-ubuntu ~ #

Inode



Python

```python
#!/usr/bin/env python
import sys

a = 42
print(sys.getrefcount(a))
b = a
print(sys.getrefcount(a))
```

Listing 2: Python

```
9
10
```

Listing 3: Output

PHP

```php
<?php
$a = 42;
xdebug_debug_zval('a');
$b = $a;
xdebug_debug_zval('a');
?>
```

Listing 4: PHP

```
a:

(refcount=1, is_ref=0),int 42

a:

(refcount=2, is_ref=0),int 42
```

Listing 5: Output

```
16 struct memoryObject{
     void* data;
18   unsigned int refCount;
   };
20
   struct memoryObject* alloc(size_t);
22 struct memoryObject* getRef(struct memoryObject*);
   void release(struct memoryObject*);
```

Listing 6: Header File

```
struct memoryObject* alloc(size_t size){
  struct memoryObject* object = 0;

  object = (struct memoryObject*) malloc(sizeof(struct
      memoryObject));

  object->refCount = 1;
  object->data = malloc(size);

  return object;
}
```

Listing 7: Allocate

```
22 struct memoryObject* getRef(struct memoryObject* object
       ){
     object->refCount++;
24   return object;
   }
```

Listing 8: Get new reference

```
28
30
32
34
36
```

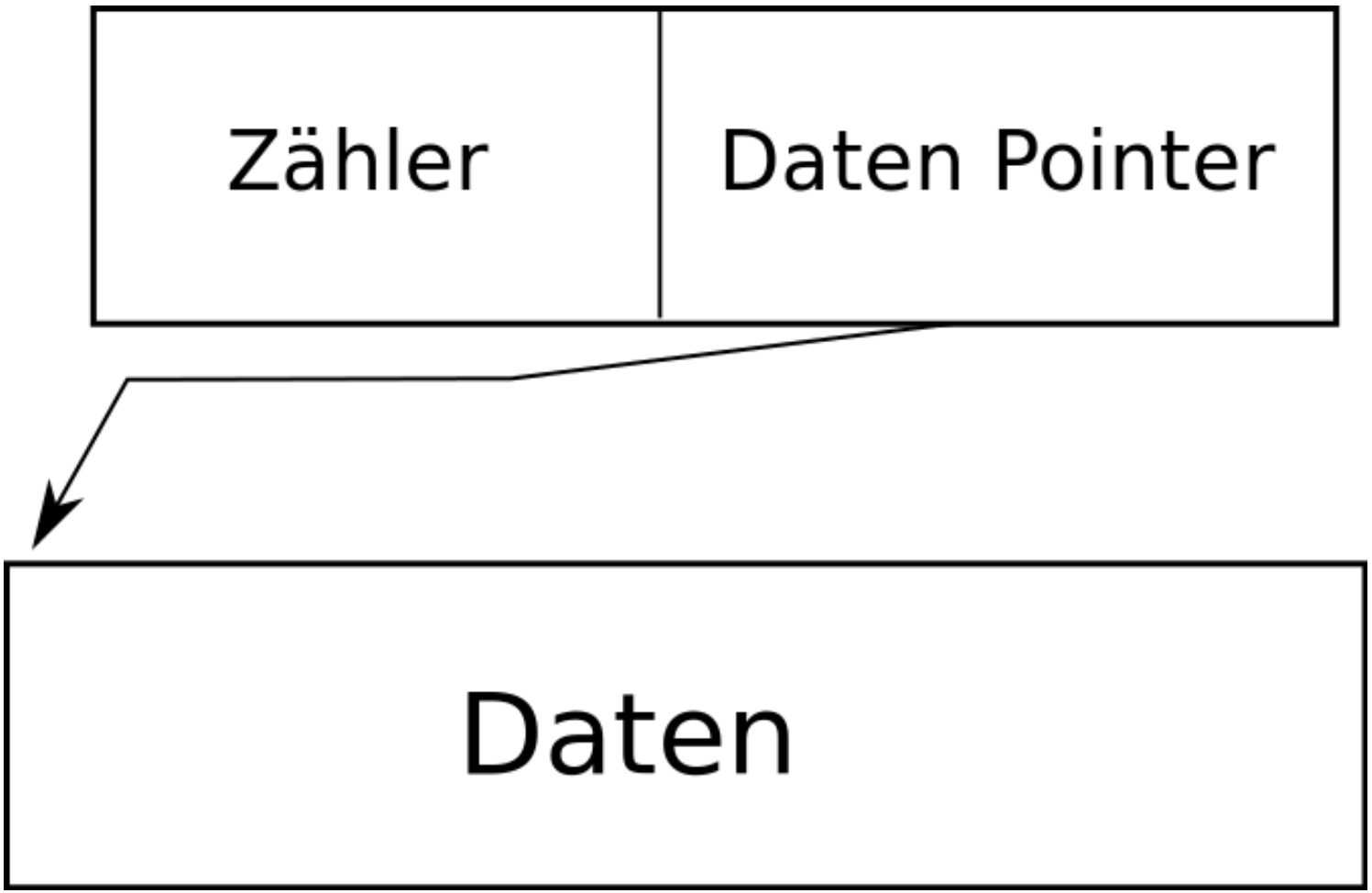
```c
void release(struct memoryObject* object){

  object->refCount--;

  if( object->refCount == 0){
    free(object->data);
    free(object);

  }
}
```

Listing 9: Release


Zähler
Daten Pointer
Daten

```
void usage(){
26   struct memoryObject* object = alloc(sizeof(int));

28   *((int*)object->data) = 5;

30   struct memoryObject* object2 = getRef(object);

32   release(object);
     release(object2);
34 }
```

Listing 10: Benutzung

```
typedef struct{
  unsigned int refCount;
  //void* data;
}memoryObject;

void* alloc(size_t);
void* getRef(void*);
void release(void*);
```

Listing 11: Header File

Speicher

Speicher

Speicher

```
void* alloc(size_t size){

  size_t overheadSize = sizeof(memoryObject);

  memoryObject* o = 0;
  char* ptr = 0;

  o = (memoryObject*) malloc(overheadSize + size);
  ptr = (char*) o;
  ptr += sizeof(memoryObject);

  o->refCount = 1;

  return (void*)ptr;
}
```

Listing 12: Allocate

```
void* getRef(void* pointer){
28
  memoryObject* o;
30  char* cptr;

32  cptr = (char*)pointer;
  cptr -= sizeof(memoryObject);
34  o = (memoryObject*)cptr;

36  o->refCount++;

38  return pointer;
}
```

Listing 13: Get new reference

```
void release(void* pointer){
  memoryObject* o;
  char* cptr;

  cptr = (char*)pointer;
  cptr -= sizeof(memoryObject);
  o = (memoryObject*)cptr;

  o->refCount--;

  if( o->refCount == 0){
    free(o);
  }
}
```

Listing 14: Release

Benutzung

```
void usage(){
28   int* intPtr = (int*)alloc(sizeof(int));

30   *intPtr = 5;

32   int* ptr2 = (int*)getRef(intPtr);

34

   release(ptr2);
36   release(intPtr);
}
```

Listing 15: Benutzung

Zyklen

```
struct link{
  void* data;
  struct link* ptr;
};

void circular(){
  struct link* ptr1 = (struct link*)alloc(sizeof(struct
      link));
  struct link* ptr2 = (struct link*)alloc(sizeof(struct
      link));

  ptr1->ptr = (struct link*)getRef(ptr2);
  ptr2->ptr = (struct link*)getRef(ptr1);

  release(ptr1);
  release(ptr2);
}
```

Listing 16: Zyklus

Zyklen

```
1  $valgrind --tool=memcheck ./ReferenceCounting
   ==12551== Memcheck, a memory error detector
3  ==12551== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
   ==12551== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
5  ==12551== Command: ./ReferenceCounting
   ==12551==
7  ==12551==
   ==12551== HEAP SUMMARY:
9  ==12551==     in use at exit: 36 bytes in 2 blocks
   ==12551==   total heap usage: 2 allocs, 0 frees, 36 bytes allocated
11 ==12551==
   ==12551== LEAK SUMMARY:
13 ==12551==    definitely lost: 36 bytes in 2 blocks
   ==12551==    indirectly lost: 0 bytes in 0 blocks
15 ==12551==      possibly lost: 0 bytes in 0 blocks
   ==12551==    still reachable: 0 bytes in 0 blocks
17 ==12551==         suppressed: 0 bytes in 0 blocks
   ==12551== Rerun with --leak-check=full to see details of leaked memory
19 ==12551==
   ==12551== For counts of detected and suppressed errors, rerun with: -v
21 ==12551== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Listing 17: Valgrind

C++

Vorteile

- Operatoren können überladen werden
- Kein manuelles zerstören

Nachteile

- Langsamer
- Nicht in C

C++

Shared Pointer

- Überladene Operatoren
- Reference Counting
- Zyklen

Benutzung

```
18 void usage(){
     shared_ptr<int> ptr1(new int);
20
     *ptr1= 10;
22
     shared_ptr<int> ptr2 = ptr1;
24 }
```

Listing 18: Shared Pointer

C++

Weak Pointer

- Erhöht die Referenz NICHT
- Kann temporär zum Strong Pointer werden
- Kann Zyklen verhindern
- Zielobjekt kann schon gelöscht sein

C++

```
void foo(){

  if(auto ptr = shared_ptr.lock()){
    bar();
  }
  else {
    load();
    bar();
  }

}
```

Listing 19: Weak Pointer

System

- Intel Core2 Quad CPU Q6600 @ 2.40GHz
- 4GB RAM DDR2 800MHz
- 64Bit
- Linux 3.5.0-21-generic
- Ubuntu 12.10
- Compiler
  - gcc (Ubuntu/Linaro 4.7.2-2ubuntu1) 4.7.2
  - OpenJDK 1.7.0_09

Programm

Optionen

- 10 000 000 Iterationen
- -O0 -lrt

Normal

```
78  for(i = 0; i < ITERATIONS; ++i){
      pointers[i] = (int*)malloc(sizeof(int));
    }
```

Listing 20: Alloc

```
82  for(i = 0; i < ITERATIONS; ++i){
      linkPtr = pointers[i];
84  }
```

Listing 21: Get Reference

```
88  for(i = 0; i < ITERATIONS; ++i){
      free(pointers[i]);
    }
```

Listing 22: Release

Normal

Typ	Zeit	Anteil
Alloc time	0.583308s	0.71
getRef time	0.032474s	0.04
Release time	0.211202s	0.26
Total time	0.826984s	

Naiv

```
46  for(i = 0; i < ITERATIONS; ++i){
      pointers[i] = (int*)alloc(sizeof(int));
48  }
```

Listing 23: Alloc

```
    for(i = 0; i < ITERATIONS; ++i){
52    linkPtr = (int*)getRef(pointers[i]);
      release(linkPtr);
54  }
```

Listing 24: Get Reference

```
    for(i = 0; i < ITERATIONS; ++i){
58    release(pointers[i]);
    }
```

Listing 25: Release

Naiv

Typ	Zeit	Anteil
Alloc time	1.139242s	0.6
getRef time	0.296284s	0.16
Release time	0.468051s	0.25
Total time	1.903577s	

Transparent

```
46  for(i = 0; i < ITERATIONS; ++i){
      pointers[i] = (int*)alloc(sizeof(int));
48  }
```

Listing 26: Alloc

```
    for(i = 0; i < ITERATIONS; ++i){
52    linkPtr = (int*)getRef(pointers[i]);
      release(linkPtr);
54  }
```

Listing 27: Get Reference

```
    for(i = 0; i < ITERATIONS; ++i){
58    release(pointers[i]);
    }
```

Listing 28: Release

Transparent

Typ	Zeit	Anteil
Alloc time	0.648253s	0.48
getRef time	0.313264s	0.23
Release time	0.381382s	0.28
Total time	1.342898s	

Shared Pointer

```
34    for(i = 0; i < ITERATIONS; ++i){
        pointers[i].reset(new int);
      }
```

Listing 29: Alloc

```
38    for(i = 0; i < ITERATIONS; ++i){
        linkPtr = pointers[i];
40    }
```

Listing 30: Get Reference

Shared Pointer

Typ	Zeit	Anteil
Alloc time	1.844750s	0.35
getRef time	2.017531s	0.39
Release time	1.356881s	0.26
Total time	5.219162s	

Java

```
12    public static void bench(){
        int i;
14      Integer linkPtr;
        Integer[] pointers = new Integer[ITERATIONS];
16
        start_time = System.nanoTime();
18      for(i = 0; i < ITERATIONS; ++i){
          pointers[i] = new Integer(0);
20      }
        alloc_time = System.nanoTime();
22
        for(i = 0; i < ITERATIONS; ++i){
24        linkPtr = pointers[i];
        }
```

Listing 31: Java

Java

Typ	Zeit	Anteil
Alloc time	6.129283253s	0.94
getRef time	0.00263494s	0.0004
Release time	0.419077866s	0.06
Total time	6.548361119s	

Normal	Naiv	Transparent	C++	Java
0.826984s	1.903577s	1.342898s	5.219162s	6.548361119s
+0%	+130%	+62%	+531%	+692%
4Byte	20Byte	8Byte	20Byte	?

Was haben wir gelernt?

- Reference Counting kann Probleme lösen...
- ...jedoch nicht alle!
- Birgt auch Probleme
- Benötigt oft Hilfe vom Benutzer
- Kein Wunderheilmittel!
- Wird in manchen Bereichen aber zwingend benötigt
- C ist am schnellsten ;)

- http://e2fsprogs.sourceforge.net/ext2intro.html
- http://www.eosgarden.com/en/articles/c-refcount/
- Boost/STL/Java/Python Dokumentation
- http://en.wikipedia.org/wiki/Reference_counting